"*Any fool can write code that machines understand, only good programmers write code that humans understand*"

Preliminary notes

- NOTE1: Each section title is to be understood with "You Should/Must" in front of it
- NOTE2: Always consider you write your code within a team of interns that do not have your skill level. They need to (i) understand (ii) modify your code

Start Cleaning Up

- Avoid Unnecessary Comparisons
  - `if (xxx == true)` => `if (xxx)`
  - **TIP/NOT DECIDED**: having 1 single `return` at end of method vs. `return` ASAP?
- Avoid (Double) Negations
  - `if (! isInOrganic())` => `if (isOrganic())`
  - Always keep positive thinking when writing conditions
- Return Boolean Expressions Directly
  - `if (a) { return true; } else { return false; }` => `return(a)`
- Simplify Boolean Expressions
  - Group into simple `isXXX()` methods
  - Use brackets to avoid having to remember boolean operators' precedence
  - Remember:
    - ♠ !A && !B == !(A || B) // true
    - ♠ !A || !B == !(A && B) // true
- Avoid NullPointerException in Conditionals
  - `if (m == null || m.isEmpty() || m.xxx) { ... }`
  - Always check !=null first
  - **TIP**: Always write the 'if's to check method arguments in the order the argument appear in the prototype (=> not forget one)
- Avoid Switch Fallthrough
  - Be careful using "switch" statement
  - Always break after each case
  - If redundant code necessary => call methods
  - If fallthrough really needed => add a comment!
  - Always write a "default" case
- Always Use Braces
  - `if (xxx) yyy => if (xxx) *{* yyy *}*`
  - Prevents errors during future code additions
  - Code indentation might make such errors very difficult to detect
- Ensure Code Symmetry
  - Group pieces of code that have the same semantics together
  - Clearly separate pieces of code that DO NOT have the same semantics

Level Up Your Code Style

- Replace Magic Numbers with Constants
  - constants are in ALL_CAPS
- Favor Enums Over Integer Constants

    - Prevents using unknown values
    - Compilation errors are cheaper to correct than runtime errors! Fail-Fast!
- Favor For-Each Over For Loops
    - Local counter is often useless and prone to mis-use and IndexOutOfBoundsExceptions
    - Works for Sets and Maps!
- Avoid Collection Modification During Iteration
    - Solution: get the Collection's iterator, and work on it
- Avoid Compute-Intense Operations During Iteration
    - Make sure that the computation-intensive operations take place as rarely as possible
    - E.g., working with regexes: `Pattern.matches()`, `String.replaceAll()` => `Pattern.compile()` + `Matcher`
- Group with New Lines
    - Visually group related code and concepts together / separate different groups from each other
    - Newspaper Metaphor: A good article starts with the title (class name), goes over section headings (public members, constructors, and methods), down to its very details (private methods)
- Favor Format Over Concatenation
    - Separate the layout of a String (how it is printed) from the data (what is being printed)
    - Use `System.out.printf()` / `String.format()` / `StringTemplate` instead of str1+str2+...
    - Add a comment with an example if the format String is not that clear
    - COMMENT: beware this is not what IDEA/Eclipse do when generating toString()!!!!
- Favor Java API Over DIY
    - Re-use Code! It's more optimized and more tested! Do not (badly) re-invent the wheel!
    - Notably static methods in Classes with final "s'. E.g., `Objects.require*`, `Collections.*`, `Arrays.*`, `Math.*`, `TimeUnit`...
    - "Knowing your API is what makes a true professional"

Use Comments Wisely

- Someone might change the code but ignore the comment => it's better have the code do enough so that comments are not necessary

- Remove Superfluous Comments
    - DO NOT repeat/rephrase what the code already says
    - Important comments are those that provide additional information
        - ♠ the "why?" of a situation
        - ♠ design choices: where another design would have been possible => why yours
    - "TODO" comments should be replaced with issues in your bug tracking system
- Remove Commented-Out Code
    - Deleted comments will not get lost if you use a proper "version control system"
- Replace Comments with Constants
    - Comments are there to explain the code. But it’s even better if the code speaks for itself!
    - Replace Magic Value~Comment by Consts or Enums with meaningful names!
- Replace Comments with Utility Methods
    - Replace "Type meaningfulName = ; return meaningfulName;" by "return computation();" + method
- Document Implementation Decisions
    - Comment Template In the context of [USE CASE], facing [CONCERN] we decided for [OPTION] to achieve [QUALITY], accepting [DOWNSIDE].
- Document Using Examples
    - Provide the general format/template + valid examples + invalid examples

    - **TIP** remember to add those examples as JUnit tests so that there's more chances the comments correspond to the code!
    - **TIP** Comments within a regular expression => See Pattern.COMMENTS
- Structure JavaDoc of Packages
    - Use as many meaningful @ annotations as possible (`@link`, NOT `@author` / `@version` / `@since` as it is in CVS)
    - Use HTML formatting
    - Provide ready-to-se examples for the main class usage
    - DO NOT repeat information that JavaDoc already generates (like list of classes)
- Structure JavaDoc of Classes and Interfaces
    - Always JavaDoc public classes!
    - Write sufficiently abstract comments so that it does not loose sync with method signatures
    - DO NOT forget invariants
    - Add Examples
    - Use HTML formatting
- Structure JavaDoc of Methods
    - They are the most important comments (e.g. used in IDEs' ToolTips)
    - Methods imply state changes and side effects. This is what must be documented
    - The JavaDoc comment must be read like a contract w.r.t these changes/effects
    - Use `@see` to link to methods with related/converse changes/effects
    - State clearly your invariants
    - Document extreme cases (null, Exceptions)
        - ♠ TIP write these cases as JUnit tests
    - Add Examples
    - Use HTML formatting + JavaDocs (`@param`, `@return`, `@throws`, `@see`)
    - Escape "<", ">" and "&"
- Structure JavaDoc of Constructors
    - AS you can't select the constructor's name, comment is even more important!
    - Explain the state of the object when the constructor finishes
    - Explain the links between the various constructors (e.g. through `@See`)

Name Things Right

- Use Java Naming Conventions
    - https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
    - com.some.package, SOME_CONSTANT, SomeClass, someMethod/someVariable
    - Use nouns for vars and verbs for methods
- Follow Getter/Setter Conventions for Frameworks
    - http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/
    - JavaBeans must have
        - ♠ a constructor without any parameters
        - ♠ correctly named *public* getter and setter for *every private* field
    - For boolean fields, the getter becomes `isXXX()`
    - As many frameworks (Hibernate, Spring, Play!, Jackson...) expect Beans, not respecting the conventions will result in compile or (very difficult to detect) runtime errors
- Avoid Single-Letter Names
    - Single letters convey little meaning, imply multiple reuse
    - Programming != Maths
    - Contextualize your code: a search for an apple is not the same as the search for a car
    - some letters/numbers (like: l & 1 and o & 0) are very difficult to distinguish
    - Most IDEs will auto-complete for you
    - Names are read much more often than they're written
    -

    - The more we use a variable/method, the more we know what it does and are able to name it correctly => Use the refactoring abilities of your IDE!
- Avoid Abbreviations
    - Use lowercased Class name only where there's no more relevant name
    - Only use very common abbreviations, like CSV
- Avoid Meaningless Terms
    - Long meaningless names can be just as burdensome as single letters
    - Remove Java-Keywords like: Interface, Abstract, Impl...
    - Remove role names like: Manager, Controller...
    - Remove generic terms like: data, info, flag (vars/methods); misc, util (packages)...
- Use Domain Terminology
    - Do not try to make your code too generic by using too generic names
    - Aligning names to the domain you're writing the code for is the best way to find balance between too short/long names

Prepare for Things Going Wrong

- Not managing Exceptions correctly makes your application appear running correctly, when it is not the case
- This results in data corruption, and the longer you take to realize the problem the more data is corrupted

- Fail Fast
    - Separate parameter validations (placed first) from the normal path (placed after)
- Always Catch Most Specific Exception
    - You should always catch the most specific exception type
    - If you catch a more general type, you risk swallowing errors that you shouldn't (e.g. NullPointerException that MUST have been fixed - thus should have crashed the programm)
    - NEVER catch Throwable, or you'll break the JVM!!!
    - **TIP** to reduce code size, Java7 introduced multiple catch blocks: `catch(NumberFormatException | IOException e)`
- Explain Cause in Message
    - The Type of the Exception only give "what" is wrong, to solve the problem the dev needs more context/details> put that info in the message
    - Give enough info so that the bug can be reproduced (thus, fixed): what was expected, what we got, context
    - BEWARE that in Web Applications leaking too much information to the user is a security breach => write detailed error messages in logs, but not in GUI
    - **TIP** reuse such error cases in JUnit tests => ensures non-regression in future code
    - Use String format rather than concatenation
- Avoid Breaking the Cause Chain
    - You will get only the highest level error, loosing all the detailed message and exact error line
    - When you rethrow an exception, always use the constructor: `Exception(String message, Throwable cause)`
    - Never skip an Exception level by forwarding the lower level cause to the higher level: +throw new XXXException(e.getCause());+
- Expose Cause in Variable
    - Remember Exception are normal Classes, you can add Fields to them
    - Providing the new error message as an Field instead of embedding it in the new error message, allows to extract it easily
    - Ensure your specific Exception class and its Fields are final, so that nobody changes the message
- Always Check Type Before Cast
    - `if (xxx instanceof XXX) { XXX xxx2 = (XXX) xxx; }`
    -

    - Otherwise you'll get a `ClassCastException` at Runtime!
    - `ClassNotFoundException` can also occur but shouldn't be caught as it reflects a Production Setup problem, not a code Problem and must make the code crash in a Development Setup
- Always Close Resources
    - Leaking resources results in DoS (the whole machine goes down!)
    - Java7 introduced the try-with-resources: `try (<resource>) {} catch(...) {...}`
    - Before Java7, use: `finally { if (res!=null) { resource.close(); }`
- Always Close Multiple Resources
    - `finally { dirStrm.close(); wrtr.close(); }` will NOT close wrtr if `dirStrm.close()` fails!!!
    - Use try-with-resources!
- Explain Empty Catch
    - An empty catch without any hint of why it's empty always looks like a bug => add a comment (CAUSE->EFFECT)
    - **TIP** You can also rename the Exception var to "ignored" to make your point more explicit

Assert Things Going Right

- TDD: write test cases BEFORE the actual code

- Structure Tests Into Given-When-Then
    - "given" = sets the stage for the actual test & captures all prerequisites
    - "when" = the operation that we actually want to test
    - "then" = asserts the result(s) that we expect from the "when"
    - Don't hesitate to explicit this structure with comments
- Use Meaningful Assertions
    - To get more detailed error messages, use assertEquals(...), not assertTrue(...)
- Expected Before Actual Value
    - BEWARE of the order of the arguments: `Assert(<expected>, <computed>)`
    - Remember to exploit the various asserts: `assertArrayEquals()`, `assertLinesMatch()`, `assertIterableEquals()`, `assertAll()`, `assertTimeout()`
- Use Reasonable Tolerance Values
    - Rounding errors can accur with doubles, expect tests to fail when doing lots of double computations
    - Preferably use `assertEquals(double expected, double actual, double delta)` with delta=0.1*10^
    - **TIP** NEVER USE FLOATING-POINT ARITHMETIC FOR MONEY, EVER!
- Let JUnit Handle Exceptions
    - Use =Assertions.assertThrows()= not `try {} catch() {}` in your tests
- Describe Your Tests
    - Name your test methods correctly as this name will be the 1rst info to appear in case of failure
    - Use `@DisplayName("what it checks")` and `@Disabled("[why it's disabled] TODO: [what's the plan to enable again]")`
- Favor Standalone Tests
    - **TIP** Use static methods to setup/init your tests rather than using `@BeforeEach` / `@BeforeAll`
    - `@BeforeEach` / `@BeforeAll` might be hidden in the Class hierarchy and are not explicitely called, thus difficut to find when you try to debug a single test
    - Each test must be understandable independently => self contained
    - Eases refactoring
- Parametrize Your Tests
    - Do not repeat asserts/write multiple tests in the same test method
        - ♠ Makes actual error more difficult to spot
        - ♠

        - Forces solving of errors sequentially
    - Create a test method with arguments together with `@ParameterizedTest()` and `@ValueSource()`
- Cover the Edge Cases
    - Edge cases are very application-dependant, but at least check type boundary values:
        - null, [],
        - "", ""
        - 0, +/-1, Integer.MIN/MAX_VALUE, Double.MIN/MAX_VALUE
    - Do not try to cover everything, but most common correct (90% tool usage)/incorrect cases (edge cases)
    - Add more tests when you encounter bugs to ensure non-regression

Design Your Objects

- Split Method with Boolean Parameters
    - If a method uses a boolean as input to switch its behaviour, replace by 2 methods with names that identify correctly each behaviour
    - Favor smaller, more reusable pieces of code, less prone to errors
- Split Method with Optional Parameters
    - Do not use null to make a parameter optional, this means your method shows several behaviours, that the prototype does not identifies
    - Split in 2 simpler methods, 1 with the parameter, 1 without the parameter
    - Not considering null an edge case is dangerous
- Favor Abstract Over Concrete Types
    - On Object declaration or method arguments or method returns, use abstract types: `List<XXX>` vs. `ArrayList<XXX>`
        - Collection: most general
        - List: order is important (`Vector`, `ArrayList`, `LinkedList`)
        - Set: no repetition (`HashSet`, `TreeSet`)
        - Map: links 2 entities (`HashMap`, `TreeMap`)
    - You will avoid many conversions from one concrete type to another
    - This will ease refactoring when switching to another concrete type
- Favor Immutable Over Mutable State
    - Misuse of objects is detected at compilation time
    - Use final when Object are not supposed to change
    - Since everything is a reference in Java, using XXX x = y creates 2 references to the same object. Modifying one will impact the other. If this was not the intent (forgot the create a copy of the instance) this will be detected at Runtime only
    - This is particularly true for "value objects" (indistinguishable if their values are equal): percentages, money, currency, times, dates, coordinates, distances...
    - **TIP** also set the Class as final, otherwise sub-classes might re-introduce mutability
- Combine State and Behavior
    - Classes containing only behavior (=Methods) but lacking state (=Fields) indicate OO-design problems (encapsulation is lost)
    - Watch out for methods that only work with their input parameters, but NOT with the Fields of their class
- Avoid Leaking References * BEWARE in your setters, if you simply copy the reference passed as argument, an external modification of the Object thazt waht passed as argumetn will also change the content of your own Object!
    - **TIP** In your setters, copy the content of the Objects/Collections passed as argument: `this.xxx = new ArrayList<>(xxxArg);`
        - Also protects against null by raising Exception if one is passed as argument!
    -

    - BEWARE in your getters, when you return an Object/Collection, even it is final (reference is not modifiable), the content can still be modified!
    - **TIP** In your getters, return an immutable Object/Collection: `return Collections.unmodifiableList(someFieldList);`
    - Not having setters at all (only constructors) makes your job much easier (not possible with some frameworks: persistence...)
- Avoid Returning Null
    - null is an edge case meaning the Object is not initialized, DO NOT use it with another meaning
    - Particularly, never return null by default in a method when you do not know what else to return
    - Solutions:
        - ♠ Throw an Exception if not having an empty/default case is a problem
        - ♠ Create a static final instance of your Class that represents the empty/default case, against which you'll be able to check equals()
        - ♠ Use Java8's Optional

Let Your Data Flow

- Favor Lambdas Over Anonymous Classes
    - Lambdas are shorter thus more readable
- Favor Functional Over Imperative Style
    - When working with collections, favor shorter functional programming style
    - Functions (with good names) allows to explicit WHAT you do over HOW you do it (Imperative paradigm)
    - Use `collection.stream()`. As a bonus they can be run in parallel (MAP/REDUCE paradigm)
    - Intermediary Operations: `map()`, `flatMap()`, `filter()`, `distinct()`, `limit()`, `skip()`, `sorted()`, [`mapToX()`, `forEach()`]...
    - Terminal Operations: `any/all/noneMatch()`, `findAny/First()`, `reduce()`, `ifPresent()`, [`collect()`, `count()`, `average()`]...
    - `Collectors.toList()`, `Collectors.groupingBy()`, `Collectors.toCollection(TreeSet::new)` ...
- Favor Method References Over Lambdas
    - Problem with Lambdas: they cannot be reused (e.g. to test hem independently)
    - => when possible write a method in your OBject and pass a Method reference (::) as argument to the stream() methods
    - **TIP** you can refer to a Constructor with ClassName::new
- Avoid Side Effects
    - Side effects (modifying outsides Objects in a Lambda) are not Thread Safe (and not Functional in spirit)
- Use Collect for Terminating Complex Streams
    - DO NOT terminate streams with forEach this generally mean you use a side effect
    - Using `collect()` / `Collector.groupingBy()` will make your code look a lot more like SQL, thus more readable
- Avoid Exceptions in Streams
    - Lambdas & Exceptions do not mix well, particularly when you need to close the resource that you are streaming/iterating onto
    - **TIP** catch the Exception inside the Lambda and return a `Stream.empty()` (might require the use of `flatMap()` to output a Stream)
- Favor Optional Over Null
    - Returning null does not force the caller to manage these edge cases, which will lead to bad crashes
    - Use return an Optional and use .ifPresent() to react correctly
- Avoid Optional Fields or Parameters
    -

    - DO NOT define a Field or Parameter as Optional<>, otherwise it can still be set to null (as Optional is an Object)!
    - Optional are only necessary in method returns
    - As you do not want the setter to be able to insert a null in your field/argument, use `Objects.requireNonNull()` to validate user input
    - If you need to set your Field to null, do it with a method so that the user can only do so in a controlled manner
    - This breaks the Java Bean convention that Getter/Setter must use the same types, and therefore might not work with some frameworks
- Use Optionals as Streams
    - An Optional IS a stream, with either zero or exactly one element.
    - You can use Stream operations like filter to act on them without managing explicitely the empty case: `.orElse()`, `orElseThrow()`

Prepare for the Real World

- Use Static Code Analysis Tools
    - Automate basic debug tasks => Helps focusing on Higher level bugs
    - Examples:
        - FindBugs/SpotBugs (text-only, no fix, false positives)
        - CheckStyle/PMD (text-only, very verbose)
        - ErrorProne (few false positive, proposes fixes)
        - SonarQube (non-free)
- Agree On the Java Format in Your Team
    - Work along agile principles (XP/SCRUM)
    - Whatever it is, define a policy (braces positions, spaces vs. tab, max line length...)
    - If you do not want to waste time creating one, use an industry standard, like https://google.github.io/styleguide/javaguide.html
    - Configure your tools to use it (IDE, Versioning tool...) https://github.com/google/google-java-format
- Automate Your Build
    - First *learn* coding Java in Text Editor + compile&run *manually*
    - Then *in production* use an IDE + manage deps&compile&test&document&deploy&run *with tools*
    - Example tools: Gradle, Maven, Ant
- Use Continuous Integration
    - Outsource tests, integration, static quality checks to dedicated machines: continuous integration servers
    - CI Tools can do that on every push to the versioning sytem
    - Example tools: Jenkins, Travis CI (cloud), Codacity (cloud)
- Prepare for and Deliver Into Production:
    - You MUST be able to respond to the question: /How long does it take you to put a change of a single line of code into production?/
    - Most CI tools will also provide means to automatically deploy
    - Prepare by collecting&monitoring debugging information: logs, metrics, dashboards, and alerts
    - Example tools: ELk-Stack, Graylog
    - Monitor Exceptions
    - Example tools: Airbrake (back end), Sentry (front end)
- Favor Logging Over Console Output
    - Writing to the console, even if buffered is extremely slow!
    - Logs have timestamps
    - Logs statements provide their line number in the code

    - Logs provide fine grained importance: FATAL / ERROR / WARNING / ... / INFO / DEBUG / ...
    - Logs size is only limited by disk space
    - Logs persist if machine is rebooted
    - Spped up things using format Strings
    - Example tools: Log4j
- Minimize and Isolate Multithreaded Code
    - First write Omni-Threaded code. Optimize it. Move to Multi-Threaded code *if measures shows it is required*
    - Really think a lot about the structure of MT code & document it thoroughly
    - Favor Immutable objects
    - Minimize and Isolate MT code in a small number of small packages
    - Use the JCIP Annotations http://jcip.net/annotations/doc/
- Use High-Level Concurrency Abstractions
    - Read Books/Docs and be sure to understand Java Memory Model & the happens-before relations between state changes
    - Use higher-level classes:
        - Semaphore, CountDownLatch, CyclingBarrier
        - AtomicInteger, LongAdder, ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue
    - Try to avoid primitive tools
        - `volatile` and `synchronized`
        - Thread#start() and the Thread#join()
        - Object#wait() and Object#notify()
- Speed Up Your Program
    - Be sure to write Streams/Lambdas without side effects and where single processing steps are independent
    - Then you can simply transform
        - `.stream().[sequential().]filter(...)` -> `.stream().parallel().filter(...)`
    - This DOES NOT work with `sort()` or `forEachOrdered()`
- Know Your Falsehoods
    - Examples of things that have tens of ways to be written/interpreted (particularly in different countries) => assume as less as possible
        - first/middle/last-names
        - emails
        - addresses/postal codes
        - CSV files
        - TimeZones

External Resources

- Books
    - Effective Java / Clean Code
    - Design Patterns: Elements of Reusable Object-Oriented Software
    - Pragmatic Unit Testing in Java 8 with JUnit
    - Continuous Integration / Release It!
    - Java Concurrency in Practice / Programming Concurrency on the JVM
    - Functional Programming in Java
    - Agile Software Development, Principles, Patterns, and Practices
- URLs
    - https://www.dzone.com
      https://dzone.com/refcardz
    - https://zeroturnaround.com

https://zeroturnaround.com/tag/cheat-sheet/
- https://www.tutorialspoint.com
- https://baeldung.com

Largely Inspired From: Java by Comparison, Simon Harrer, Jörg Lenhard, Linus Dietz.